Naval Research Laboratory
Washington, DC 20375-5320

An Implementation of the Pump: The Event Driven Pump

BRUCE MONTROSE
MYONG H. KANG

Center for High Assurance Computer Systems
Information Technology Division

DTIC QUALITY INSPECTED 4

May 21, 1996

19960606 119

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget. Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE May 21, 1996	3. REPORT TYPE AND DATES COVERED	
4. TITLE AND SUBTITLE An Implementation of the Pump: The Event Driven Pump			5. FUNDING NUMBERS PE - 33401G
6. AUTHOR(S) Bruce Montrose and Myong H. Kang			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory Washington, DC 20375-5320			8. PERFORMING ORGANIZATION REPORT NUMBER NRL/MR/5540--96-7850
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Security Agency 9800 Savage Road Ft. Meade, MD 20770			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

As computer systems become more open and interconnected, the need for reliable and secure communication also increases. In this report, we discuss a communication device, the NRL Pump, and introduce an implementation of the Pump: the Event Driven Pump that balances the requirements of reliability and security. The Pump provides acknowledgements (Acks) to the message source to insure reliability. These Acks are also used for flow control to inhibit the Pump's buffer from becoming or staying full. This is desirable because once the buffer is filled there exists a huge covert communication channel.

We have prepared this report for system designers and programmers who want to understand the basic structure of the event-driven Pump. We also hope this report is helpful to the people who will maintain the Pump Code. In this report, we assume that the reader is familiar with the material presented in [2] and [3].

14. SUBJECT TERMS			15. NUMBER OF PAGES
NRL pump Secure communications	Covert channels		26
			16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

CONTENTS

An Implementation of the Pump: The Event Driven Pump

Abstract

As computer systems become more open and interconnected, the need for reliable and secure communication also increases. In this report, we discuss a communication device, the NRL Pump, and introduce an implementation of the Pump: the Event Driven Pump that balances the requirements of reliability and security. The Pump provides acknowledgments (Acks) to the message source to insure reliability. These Acks are also used for flow control to inhibit the Pump's buffer from becoming or staying full. This is desirable because once the buffer is filled there exists a huge covert communication channel.

We have prepared this report for system designers and programmers who want to understand the basic structure of the event-driven Pump. We also hope this report is helpful to the people who will maintain the Pump code. In this report, we assume that the reader is familiar with the material presented in [2] and [3].

1 Introduction

Sharing information between computer systems is undeniably the wave of the future. As computer systems become more open and distributed, the security concerns relating to information exchange between different systems will grow.

Two security concerns are paramount:

1. There should be no intrusion of unauthorized entities. Research on access control, virus detection and prevention, etc. focus on this aspect of security.

2. No unauthorized information flow between computer systems.

In this report, we address the second concern—security means no unauthorized information flows. In a multilevel secure system information from Low should be able to be passed to High but information from High is prohibited to be passed down to Low (i.e., the Bell-LaPadula requirements (BLP) [1, 8]). Hence, if two systems, which are at different security levels, are connected then a mechanism is needed to guarantee that no information flows down.

At first glance, there seems to be no problem meeting the BLP requirements. Simply design a system so that information only flows up, not down (i.e., read-down and blind write-up methods [1]). However, secure communication mechanisms should also satisfy functional requirements (e.g., reliability, performance).

When Low sends a message, the Pump provides an Ack to Low. This Ack gives us the assurance that our message will be safely delivered to High. Without Acks how does Low know that High is ready to receive messages? How does Low know that a message arrived at High? The Acks also provide flow control capability.

However, the propagation of Acks from High to Low violates BLP. Hence we should carefully analyze the capacity, in Shannon's sense, of the potential information flow from High to Low. This analysis was presented in [3].

A brief description of the basic Pump, one sender/Low and one receiver/High, is presented in section two. An application of the Pump in secure networks or more dynamic system configuration, using the foundations developed here, is dealt with in separate papers[4, 5].

2   The Basic Pump

A secure one-way communication device should satisfy two equally important types of requirements: the security requirements (e.g., no information flow from High to Low) and the functional requirements (e.g., reliability and performance). Even though blind write-up and the similar read-down methods may satisfy the security requirements [2, 3], they do not satisfy the functional requirements.

Let us discuss a problem inherent in blind write-up under the optimistic assumption of error-free message transmission. Low sends messages to High, and these messages reside in a (intermediate) buffer until High can receive them. If Low sends messages too fast or, similarly, has periods of extreme burstiness [7] the buffer can fill. Blind write-up sends no Acks back to Low, hence Low has no way of knowing that the buffer is full; therefore under the blind write-up paradigm messages will be dropped and never reach High. Therefore, we need some feedback from High to Low. However, if this feedback is under the control of High, then the feedback can be used as a covert channel. Hence the feedback must be modified. This is what the Pump[1] does.

Kang and Moskowitz introduced the (NRL) basic Pump as a device that balances all

---

[1]The term Pump, without any further modification, refers to the theory behind the basic, network, and generalized Pumps.

requirements [2, 3]. Note that no device can totally satisfy all requirements [6]. The basic Pump can handle only one sender and one receiver. An abstract view of the basic Pump is as follows:
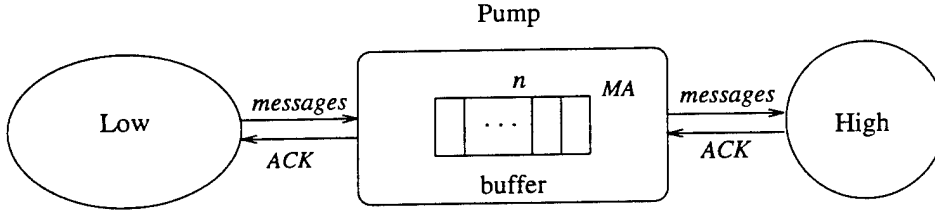

Figure 1: The basic Pump

The basic Pump places a non-volatile buffer (size n) between Low (the transmitter/input) and High (the receiver/output). The Pump sends Acks to Low at probabilistic times based upon a moving average of the past m High Ack times [2, 3]. A High Ack time is the time from when the Pump sends a message to High to the time when the Pump receives an Ack from High. Low uses a handshake protocol and does not send a new message until a previous message has been Acked.

If we consider the basic Pump to be at the high level, the Acks from the basic Pump to Low do violate the security requirements. However, the basic Pump is designed so that even though the security requirements are violated the covert channel capacity can be kept within specified bounds (as small as desired) while still satisfying the functionality requirements.

In brief, the basic Pump balances requirements as follows:

- The basic Pump is reliable, unlike blind write-up, because it sends an Ack to Low in response to each message from Low. Further, Low waits for an Ack before sending its next message (handshake protocol).

- The basic Pump impacts performance minimally because:

  1. If High's Ack rate is faster than Low's message input rate then the basic Pump does not add any random delay.

  2. If High's Ack rate is slower than Low's message input rate then the basic Pump adds random delay to slow down Low, so that Low's input rate is approximately the same as the High's Ack rate (service rate).

  Since, in general, the input rate cannot exceed the service rate the basic Pump does not hurt performance.

- The controlling of the Ack time to Low is the key to controlling covert channel capacity. This scheme keeps the buffer from filling up. If the buffer were to become full, it would open up a huge covert channel, the full buffer channel [3], which the basic Pump avoids. The basic Pump's covert channel capacity can be controlled by the sizes of the buffer (n) and how many past High Ack times are used to compute the moving average (m) [3].

3 Implementation—Event Driven Pump

In this section we present an implementation of the basic Pump which we call the Event Driven Pump. We will discuss the chosen development platform, software architecture, and finally we will present the pseudo code which was used to build the working version of the Event Driven Pump.

3.1 Development Platform

The XTS-300 was chosen to implement the Event Driven Pump because of its (1)—availability, and (2)—its operating system (STOP 4.1) has a B3 rating. The use of a certified operating system preempts the need for certifying the system services (e.g., interprocess communication, file-system, etc.) used by the Event Driven Pump.
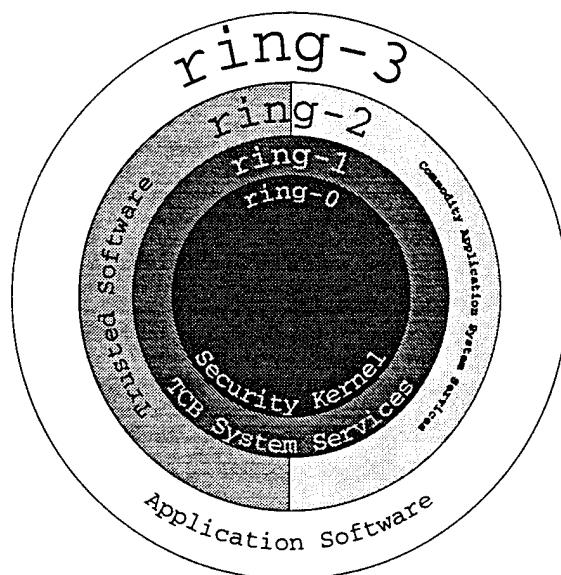

Figure 2: STOP security ring structure

The security kernel provides basic system operating services (e.g., resource management, process scheduling, interrupt, trap handling) and enforcement of system security (e.g., security and integrity rules). Privileged software known as Trusted Software provides additional security services outside the kernel. Commodity Application System Service (CASS) provides untrusted operating system services to application programs on the XTS-300 [2].

The XTS-300 supports trusted processes. A process is trusted if the process has privileges that exempt it from specific access control rules (e.g., no read-up or no write-down rule). Since the basic Pump sends Acks back to Low, a portion of the basic Pump must have trusted processes.

---

[2] This a B3 rated machine from Wang (HFSI).

3.2 Software Architecture

Ideally the entire Event Driven Pump should be implemented as trusted ring-2 processes. Since ring-2 processes on the XTS-300 currently cannot access TCP/IP, it was necessary to have portions of the Pump that needed to access TCP/IP to be implemented in ring-3. The only mechanism available for ring-3 and ring-2 processes to communicate on the XTS-300 is the file-system. We used special files with a first-in-first-out protocol called FIFOs as the communication channel between the ring-2 and ring-3 components of the Event Driven Pump.

We conceptually refer to each component of the Event Driven Pump as an Object. Each Object was implemented as a separate process designed to accomplish a specific task when certain events occurred. An event is either a message sent by another Object or the completion of an I/O operation. Inter-Object message passing was accomplished via the Interprocess Communication (IPC) interface provided by the operating system.

Two store and forward buffers (SAFB) were used; one in volatile shared memory and the other is in non-volatile disk storage. All ring-2 Objects have access to both SAFBs. The non-volatile SAFB was required for recovery purposes in the event the XTS-300 should halt during operation. The volatile SAFB serves as the I/O buffer for data transferred to the non-volatile SAFB, the Output FIFO, and the Input FIFO.

In this paper we refer to the packets of data that are propagated through the Pump as *messages*. A message is composed of a length field specifying the number of bytes in the message, and the actual message content. These messages, along with a checksum field are stored in the SAFB as *records*. This checksum field is computed from the content of the message and is needed in order to have reliable recovery procedures in place. The SAFB has a fixed size which is determined by the maximum queue size and the maximum length of a message; Both of which are configuration parameters. Each SAFB record has a unique ID from 0 to MAXRECID. The location of a record within the SAFB is easily computed from the ID since the records have a fixed size. Note that the fixed record size merely imposes a maximal length for messages and does *not* require messages to be a fixed size. An *occupied* flag is associated with each record to indicate a record's availability. After a message has been successfully propagated through the Pump the occupied flag in the associated record is set to FALSE so that the record may be used by another message. Appended to the end of the non-volatile SAFB are two pointers containing the message IDs of the last Acked records on the low and high side of the Pump respectively. After a system failure has occurred the recovery process will use this information to help establish a recovery order for un-delivered records.
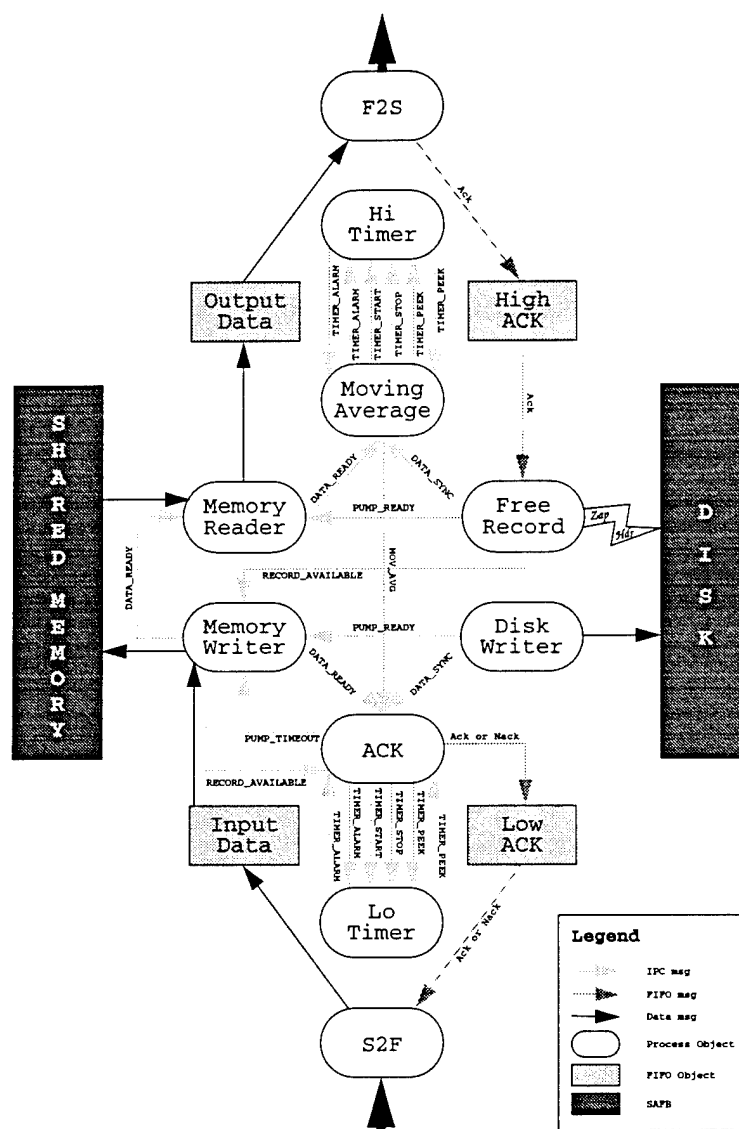
Figure 3: The Event Driven Pump

A brief description of each Object is as follows. The pseudo code which is presented in the following section has a more detailed description of each Object's functionality.

S2F   It delivers data from the Low Client Application (LCA) to the Input Data FIFO and relays the Pump's Ack from the Low Ack FIFO to LCA.

Memory Writer   It reads data from the Input Data FIFO, stores the data into Shared Memory, and sends IPC messages to the Disk Writer and Memory Reader Objects informing them that there is data to be processed. It also sends an IPC message to the Ack Object so that it can keep track of timing (e.g., time-out, disk writing overhead).

Memory Reader   It reads data from Shared Memory and writes the data to the Output Data FIFO. It also sends an IPC message to the Moving Average Object to trigger the moving average computation.

Ack   It computes the randomized delay based on the moving average and sends an Ack to the Low Ack FIFO after data is safely stored in the non-volatile SAFB.

Lo Timer   It is the timer for the Ack Object.

Moving Average   It computes the moving average and sends the updated moving average to the Ack Object.

Hi Timer   It is the timer for the Moving Average Object.

Disk Writer   It writes data to the disk and sends an IPC message to the Ack Object which indicates that the data is safely stored in the non-volatile SAFB.

Free Record   It reads an Ack from the High Ack FIFO, sends an IPC message to the Moving Average Object so that the new moving average can be computed, and removes the data from the disk. After it frees the buffer space it sends an IPC message to the Memory Writer Object to indicate the space is now available.

F2S   It delivers data from the Output Data FIFO to the High Client Application (HCA) and sends Acks to the High Ack FIFO upon successful delivery of the data to the HCA.

Not shown in Figure 3 are two additional processes called the Recovery Object and the Create Pump Object. The Recovery Object is responsible for the orderly recovery from system failure. The Create Pump Object sets up all data structures and creates all the other Objects described above including the Recovery Object. Unlike the other Objects described above, the Recovery Object and Create Pump Object will terminate after their tasks have been completed.

3.3 Pseudo Code

Here we present the Pseudo Code for all the Objects of the Event Driven Pump. The code was meant to convey enough detail so that system designers could validate conceptual models. The C-like syntax was chosen since the C programming language would be used during implementation. Many abstract data types such as "Access" were left undefined since their definitions are irrelevant to the understanding of the concepts being presented here.

Create Pump Object

The Create Pump Object is firstly responsible for setting up and initializing all resources required by an operational Event Driven Pump. Secondly, the Create Pump Object must activate all process Objects that make up the Event Driven Pump. Finally, the Create Pump Object begins the transaction cycles of both the low and high interface of the Pump.

```
Local Storage:
    RecordID    ID;
    Event       event;
    Access      lo_access, hi_access;
    int         output_fifo, input_fifo;
    PumpInfo    pump_info;
DeterminePortAccess ( &lo_access, &hi_access );
SetupFIFOs ( lo_access, hi_access );
SetupSharedMemory();
CreateDiskBuffer();
RegisterEvents ( PUMP_READY_EVENT );
RegisterObject ( CREATE_PUMP_OBJECT );
LoadProcess ( F2S_PROGRAM, hi_access );
LoadProcess ( S2F_PROGRAM, lo_access );
LoadObject ( RECOVERY_OBJECT );
LoadObject ( MEMORY_WRITER_OBJECT );
LoadObject ( MEMORY_READER_OBJECT );
LoadObject ( DISK_WRITER_OBJECT );
LoadObject ( FREE_RECORD_OBJECT );
LoadObject ( LO_TIMER_OBJECT );
LoadObject ( HI_TIMER_OBJECT );
LoadObject ( ACK_OBJECT );
LoadObject ( MOVING_AVERAGE_OBJECT );
AwaitEvent ( &event, PUMP_READY_EVENT ); /* sent by RECOVERY_OBJECT */
SignalEvent ( MEMORY_WRITER_OBJECT, PUMP_READY_EVENT );
SignalEvent ( MEMORY_READER_OBJECT, PUMP_READY_EVENT );
```

```
UnRegisterObject ( CREATE_PUMP_OBJECT );


GetPumpInfo ( &pump_info );
WriteFile ( input_fifo, &pump_info, sizeof(PumpInfo) );
WriteFile ( output_fifo, &pump_info, sizeof(PumpInfo) );
CloseFile ( output_fifo );
```

Recovery Object

The task of the Recovery Object is to scan the non-volatile SAFB looking for records with a non-zero length specification and correct checksum values. A non-zero length record with an incorrect checksum occurs when a system halt prevented the Pump from completing the transfer from volatile SAFB to non-volatile SAFB. After determining which records are to be recovered, the Recovery Object writes the corresponding messages to the Output Data FIFO where the F2S Object will be expecting them. The length of each record is set to zero after it has been recovered.

```
Global Storage:
    Record        Rec[MAXRECORDS];
Local Storage:
    Char          notify;
    Integer       input_fifo,output_fifo,recover_count;
    RecordID      ID;
    RecordHeader  hdr;
RegisterObject ( RECOVERY_OBJECT );
OpenDiskBuffer();
recover_count = 0;
DoForAll ( ID = 0 to MAXRECID )
    SeekRecord ( ID );
    ReadRecordHeader ( ID );
    ReadRecordData ( ID );
    If ( Rec[ID].hdr.occupied == TRUE ) Then
        If ( Rec[ID].hdr.CRC == RecordChecksum(ID) ) Then
            ++recover_count;
        Else
            UnsetOccupiedFlag ( ID );
            SignalEvent ( MEMORY_WRITER_OBJECT, RECORD_AVAILABLE_EVENT, ID );
        EndIf
    Else
        SignalEvent ( MEMORY_WRITER_OBJECT, RECORD_AVAILABLE_EVENT, ID );
    EndIf
```

```
EndForAll
input_fifo  = OpenFile ( INPUT_FIFO );
output_fifo = OpenFile ( OUTPUT_FIFO );
hdr.reclen = 0;
ID = GetLastHiRecordID();
If ( ID != NO_RECID ) Then
    WriteFile ( output_fifo, &Rec[ID].hdr.reclen, sizeof(Rec[ID].hdr.reclen) );
    WriteFile ( output_fifo, Rec[ID].dta, Rec[ID].hdr.reclen );
Else
    WriteFile ( output_fifo, &hdr.reclen, sizeof(hdr.reclen) );
EndIf
ID = GetLastLoRecordID();
If ( ID != NO_RECID ) Then
    WriteFile ( input_fifo, &Rec[ID].hdr.reclen, sizeof(Rec[ID].hdr.reclen) );
    WriteFile ( input_fifo, Rec[ID].dta, Rec[ID].hdr.reclen );
Else
    WriteFile ( input_fifo, &hdr.reclen, sizeof(hdr.reclen) );
EndIf
CloseFile ( output_fifo );
CloseFile ( input_fifo );
If ( recover_count > 0 ) Then
    ID = (ID + 1) mod MAXRECORDS;
    DoWhile ( Rec[ID].hdr.occupied == FALSE )
        ID = (ID + 1) mod MAXRECORDS;
    EndWhile
    SignalEvent ( FREE_RECORD_OBJECT, PUMP_READY_EVENT, ID );
    DoWhile ( Rec[ID].hdr.occupied == TRUE )
        SignalEvent ( MEMORY_READER_OBJECT, DATA_READY_EVENT, ID );
        ID = (ID + 1) mod MAXRECORDS;
    EndWhile
Else
    SignalEvent ( FREE_RECORD_OBJECT, PUMP_READY_EVENT, 0 );
EndIf
SignalEvent ( CREATE_PUMP_OBJECT, PUMP_READY_EVENT );
UnRegisterObject ( RECOVERY_OBJECT );
```

Memory Writer Object

The task of the Memory Writer Object is to secure an available record, to read a message from the Input Data FIFO into the volatile SAFB, and to inform both the Memory Reader

Object and the Disk Writer Object when the resulting SAFB record is ready to be processed. The Memory Writer Object also informs the Ack Object when to begin the timing of the Low delivery rate.

```
Global Storage:
    Record          Rec[MAXRECORDS];
Local Storage:
    Event           event;
    RecordID        ID;
    Integer         input_fifo;
    Record          rec;
    Boolean         timeout;
RegisterEvents (  PUMP_READY_EVENT,
                  PUMP_TIMEOUT_EVENT,
                  RECORD_AVAILABLE_EVENT );
RegisterObject ( MEMORY_WRITER_OBJECT );
input_fifo = OpenFile ( INPUT_FIFO );
timeout = FALSE;
DoForever
    If ( timeout == FALSE ) Then
        AwaitEvent ( &event, PUMP_READY_EVENT );
        DiscardEvent ( PUMP_TIMEOUT_EVENT );
    Else
        timeout = FALSE;
    EndIf
    ReadFile ( input_fifo, &rec.hdr.reclen, sizeof(rec.hdr.reclen) );
    SignalEvent ( ACK_OBJECT, DATA_READY_EVENT );
    AwaitEvent ( &event, ANY_REGISTERED_EVENT );
    Case ( event.type )
        RECORD_AVAILABLE_EVENT:
            ID = event.recID;
            SignalEvent ( ACK_OBJECT, RECORD_AVAILABLE_EVENT );
            ReadFile ( input_fifo, Rec[ID].dta, rec.hdr.reclen );
            Rec[ID].hdr.reclen = rec.hdr.reclen;
            SignalEvent ( DISK_WRITER_OBJECT, DATA_READY_EVENT, ID );
            SignalEvent ( MEMORY_READER_OBJECT, DATA_READY_EVENT, ID );
        PUMP_TIMEOUT_EVENT:
            ReadFile ( input_fifo, rec.dta, rec.hdr.reclen );
            timeout = TRUE;
    EndCase
```

```
EndForever
```

Memory Reader Object

The task of the Memory Reader Object is to transfer records from the volatile SAFB into the Output Data FIFO as messages, and to let the Moving Average Object know when to begin timing the high delivery rate.

```
Global Storage:
    Record          Rec[MAXRECORDS];
Local Storage:
    Event           event;
    Integer         output_fifo;
    RecordID        ID;
RegisterEvents ( PUMP_READY_EVENT, DATA_READY_EVENT );
RegisterObject ( MEMORY_READER_OBJECT );
output_fifo = OpenFile ( OUTPUT_FIFO );
DoForever
    AwaitEvent ( &event, PUMP_READY_EVENT );
    If ( event.value == ACK_OK ) Then
        AwaitEvent ( &event, DATA_READY_EVENT );
        ID = event.recID;
    EndIf
    SignalEvent ( MOVING_AVERAGE_OBJECT, DATA_READY_EVENT );
    WriteFile ( output_fifo, &Rec[ID].hdr.reclen, sizeof(Rec[ID].hdr.reclen) );
    WriteFile ( output_fifo, Rec[ID].dta, Rec[ID].hdr.reclen );
EndForever
```

Disk Writer Object

The task of the Disk Writer Object is to transfer a record from the volatile SAFB into the non-volatile SAFB and to inform the Ack Object that a low delivery has been completed after the transfer has finished. It is the Disk Writer Object which actually computes the checksums stored in the non-volatile SAFB.

```
Global Storage:
    Record          Rec[MAXRECORDS];
Local Storage:
    Event           event;
    RecordID        ID;
RegisterEvents ( DATA_READY_EVENT );
RegisterObject ( DISK_WRITER_OBJECT );
OpenDiskBuffer();
DoForever
    AwaitEvent ( &event, DATA_READY_EVENT );
    ID = event.recID;
    Rec[ID].hdr.occupied = TRUE;
    Rec[ID].hdr.CRC = RecordChecksum ( ID );
    RecordSeek ( ID );
    WriteRecordHeader ( ID );
    WriteRecordData ( ID );
    SetLastLoRecordID ( ID );
    SignalEvent ( ACK_OBJECT, DATA_SYNC_EVENT );
    SignalEvent ( MEMORY_WRITER_OBJECT, PUMP_READY_EVENT );
EndForever
```

Free Record Object

The task of the Free Record Object is to await Acks from the High Ack FIFO, to write a zero length specification into the record for which the Ack applies, and to inform the Memory Writer Object of the availability of this record.

```
Local Storage:
    Event           event;
    RecordID        ID;
    Integer         hiack_fifo;
    AckCode         acode;
RegisterEvents ( PUMP_READY_EVENT );
RegisterObject ( FREE_RECORD_OBJECT );
hiack_fifo = OpenFile ( HIACK_FIFO );
```

```
OpenDiskBuffer();
AwaitEvent ( &event, PUMP_READY_EVENT ); /* RECOVERY_OBJECT will send */
ID = event.recID;
DoForever
    ReadFile ( hiack_fifo, &acode, sizeof(AckCode) );
    If ( acode == ACK_OK ) Then
        WriteNullRecordHeader ( ID );
        SetLastHiRecordID ( ID );
        SignalEvent ( MEMORY_WRITER_OBJECT, RECORD_AVAILABLE_EVENT, ID );
    EndIf
    SignalEvent ( MOVING_AVERAGE_OBJECT, DATA_SYNC_EVENT );
    SignalEvent ( MEMORY_READER_OBJECT, PUMP_READY_EVENT, ID, acode );
    If ( acode == ACK_OK ) Then
        ID = (ID + 1) mod MAXRECORDS;
    EndIf
EndForever
```

Moving Average Object

The task of the Moving Average Object is to compute the moving average of the high delivery
rate and to relay this value to the Ack Object so that it may adjust its Low Ack rate accordingly
(flow control).

```
Local Storage:
    Event       event;
    Integer     Hm;
    Boolean     timeout;
RegisterEvents ( DATA_READY_EVENT,
                 DATA_SYNC_EVENT,
                 TIMER_PEEK_EVENT,
                 TIMER_ALARM_EVENT );
RegisterObject ( MOVING_AVERAGE_OBJECT );
SetEventHandler ( TIMER_ALARM_EVENT, AlarmHandler );
DoForever
    If ( timeout == FALSE ) Then
        AwaitEvent ( &event, DATA_READY_EVENT );
    EndIf
    SignalEvent (HI_TIMER_OBJECT, TIMER_START_EVENT);
    SignalEvent (HI_TIMER_OBJECT, TIMER_ALARM_EVENT,
                 MOVING_AVERAGE_OBJECT, PUMP_TIMEOUT_VALUE);
    timeout = FALSE;
```

```
      AwaitEvent ( &event, DATA_SYNC_EVENT, PUMP_TIMEOUT_VALUE );
   If ( timeout ) Then
      Hm = MovingAverage ( PUMP_TIMEOUT_VALUE );
   Else
      SignalEvent ( HI_TIMER_OBJECT, TIMER_PEEK_EVENT, MOVING_AVERAGE_OBJECT );
      AwaitEvent ( &event, TIMER_PEEK_EVENT );
      If ( event.value > PUMP_TIMEOUT_VALUE ) Then
         event.value = PUMP_TIMEOUT_VALUE;
      EndIf
      Hm = MovingAverage ( event.value );
      SignalEvent ( HI_TIMER_OBJECT, TIMER_STOP_EVENT );
   EndIf
   SignalEvent ( ACK_OBJECT, MOV_AVG_EVENT, Hm );
EndForever


AlarmHandler()
   timeout = TRUE;
   return;
```

Ack Object

The task of the Ack Object is to insure that an Ack or a Nack is written to the Low Ack FIFO before the designated Pump timeout period, to insure that the Low Ack rate has a modified exponential distribution with a mean equivalent to the mean High Ack rate, and to inform the Memory Writer Object should a Pump timeout occur. A Pump timeout will occur when there are no available records in the SAFB.

```
Local Storage:
   Event       event;
   Boolean     timeout;
   Integer     Si, compute_time, time_left, lag_time, loack_fifo;
   Floating    Mu, Hm, Di, R1, R2, R3;
   AckCode     acode;
RegisterEvents ( MOV_AVG_EVENT,  .
                 TIMER_ALARM_EVENT,
                 TIMER_PEEK_EVENT,
                 DATA_READY_EVENT,
                 DATA_SYNC_EVENT,
                 RECORD_AVAILABLE_EVENT );
RegisterObject ( ACK_OBJECT );
SetEventHandler ( MOV_AVG_EVENT, MovAvgHandler );
```

```
SetEventHandler ( TIMER_ALARM_EVENT, AlarmHandler );
timeout = FALSE;
loack_fifo = OpenFile ( LOACK_FIFO );
DoForever
    If ( timeout == TRUE ) Then
        SignalEvent ( LO_TIMER_OBJECT, TIMER_STOP_EVENT );
        SignalEvent ( MEMORY_WRITER_OBJECT, PUMP_TIMEOUT_EVENT );
        acode = ACK_RESEND;
        WriteFile ( loack_fifo, &acode, sizeof(acode) );
        timeout = FALSE;
    EndIf
    AwaitEvent ( &event, DATA_READY_EVENT );
    SignalEvent ( LO_TIMER_OBJECT, TIMER_START_EVENT );
    SignalEvent ( LO_TIMER_OBJECT, TIMER_ALARM_EVENT,
                    ACK_OBJECT, PUMP_TIMEOUT_VALUE );
    AwaitEvent ( &event, RECORD_AVAILABLE_EVENT, PUMP_TIMEOUT_VALUE );
    If ( timeout == TRUE ) Continue;
    SignalEvent ( LO_TIMER_OBJECT, TIMER_PEEK_EVENT, ACK_OBJECT );
    AwaitEvent ( &event, TIMER_PEEK_EVENT );
    If ( timeout == TRUE ) Continue;
    lag_time = event.value;
    time_left = PUMP_TIMEOUT_VALUE - lag_time;
    AwaitEvent ( &event, DATA_SYNC_EVENT, time_left );
    If ( timeout == TRUE ) Continue;
    SignalEvent ( LO_TIMER_OBJECT, TIMER_ALARM_EVENT, ACK_OBJECT );
    SignalEvent ( LO_TIMER_OBJECT, TIMER_PEEK_EVENT, ACK_OBJECT );
    AwaitEvent ( &event, TIMER_PEEK_EVENT );
    Si = event.value;
    time_left = PUMP_TIMEOUT_VALUE - Si;
    Mu = Hm - Si;
    If ( Mu <= 0 ) Mu = EPSILON;
    R1 = RandomExp ( Mu, PUMP_TIMEOUT_VALUE );
    If ( LagTime < EPSILON or Si >= Hm ) Then
        Di = (R1 > excess_time) ? excess_time : R1;
    Else
        R2 = Random ( 0, PUMP_TIMEOUT_VALUE - Hm );
        If ( R1 <= R2 ) Then
            Di = R1;
        Else
```

```
            R3 = Random ( R2 + Hm, PUMP_TIMEOUT_VALUE );
            Di = R3 - Si;
        EndIf
    EndIf
    SignalEvent ( LO_TIMER_OBJECT, TIMER_PEEK_EVENT, ACK_OBJECT );
    AwaitEvent ( &event, TIMER_PEEK_EVENT );
    compute_time = event.value - Si;
    time_left = PUMP_TIMEOUT_VALUE - event.value;
    delay_time = Di - compute_time;
    If ( delay_time >= time_left ) delay_time = time_left - lag_time;
    If ( delay_time >= 1 ) Sleep ( delay_time );
    SignalEvent ( LO_TIMER_OBJECT, TIMER_STOP_EVENT );
    timeout = FALSE;
    acode = ACK_OK;
    WriteFile ( loack_fifo, &acode, sizeof(AckCode) );
EndForever


AlarmHandler()
    timeout = TRUE;
    return;


MovAvgHandler(event)
    Hm = event.value;
    return;
```

Timer Object

The task of the Timer Object is to function as a stop watch used by both the Ack Object and the Moving Average Object to time Low and High delivery rates respectively. The Timer Object also provides an alarm function whereby a client Object can specify an alarm value and then be notified via an alarm event when the specified alarm time interval has elapsed. This same Object is used for both the Low Timer Object and the High Timer Object.

```
Local Storage:
    Event       event;
    Boolean     stop;
    Integer     timer, alarm, alarm_object;
SetEventHandler ( TIMER_STOP_EVENT, TimerStopHandler );
SetEventHandler ( TIMER_PEEK_EVENT, TimerPeekHandler );
SetEventHandler ( TIMER_ALARM_EVENT, TimerAlarmHandler );
DoForever
```

```
      AwaitEvent ( &event, TIMER_START_EVENT );
      stop = FALSE;
      timer = 0;
      alarm = 0;
      DoWhile ( stop == FALSE )
         SuspendExecution ( TIMER_INTERVAL );
         timer = timer + 1;
         If ( alarm > 0 && timer >= alarm ) Then
            SignalEvent ( alarm_object, TIMER_ALARM_EVENT, timer );
            alarm = 0;
         EndIf
      EndWhile
EndForever


TimerStopHandler ()
   stop = TRUE;
   return;


TimerPeekHandler ( Event event )
   SignalEvent ( event.value, TIMER_PEEK_EVENT, timer );
   return;


TimerAlarmHandler ( Event event )
   alarm = event.value;
   If ( alarm > 0 ) alarm_object = event.value;
   return;
```

F2S Object

The task of the F2S Object is to read a message from the output FIFO, and relay the message to a socket which is used by a High application expecting to read data from the Pump, to write an Ack to the High Ack FIFO after the High application receives the data, and to remember the last message that was Acked from the High application. When a High application initiates communication with the F2S Object, the F2S Object must send some Pump information back to the High application. This Pump information should include Pump version, maximum queue size, maximum message length, and the last message that was Acked.

```
Local Storage:
   AckCode   acode;
   Record    record;
   Integer   output_fifo, hiack_fifo;
```

```
    Integer  hitcpip_socket;
    PumpInfo pump_info;
    Boolean  have_data;
output_fifo  = OpenFile ( OUTPUT_FIFO );
hiack_fifo   = OpenFile ( HIACK_FIFO );
hitcpip_socket = OpenSocket ( HITCP_SOCKET );
ReadFile ( output_fifo, &pump_info, sizeof(PumpInfo) );
DoForever
    AwaitConnection ( hitcpip_socket );
    WriteFile ( hitcpip_socket, &pump_info, sizeof(PumpInfo) );
    acode = ACK_OK;
    have_data = FALSE;
    DoWhile ( Connected(hitcpip_socket) )
        If ( acode == ACK_OK ) Then
            If ( have_data == TRUE ) Then
                WriteFile ( hiack_fifo, &acode, sizeof(AckCode) );
                pump_info.record = record;
            EndIf
            ReadFile ( output_fifo, &record.hdr.reclen, sizeof(Integer) );
            ReadFile ( output_fifo, record.dta, record.hdr.reclen );
            acode = ACK_RESEND;
            have_data = TRUE;
        Else
            WriteFile ( hitcpip_socket, &record.hdr.reclen, sizeof(Integer) );
            WriteFile ( hitcpip_socket, record.dta, record.hdr.reclen );
            ReadFile ( hitcpip_socket, &acode, sizeof(AckCode) );
        EndIf
    EndWhile
EndForever
```

S2F Object

The task of the S2F Object is to read messages from the socket used by the Low application and relaying those messages into the Input Data FIFO, and to read and relay the Acks that come back via the Low Ack FIFO. The S2F Object must keep track of the last Acked message so that it may send it along with other Pump information after a socket connection has been established with the Low application.

```
Local Storage:
    AckCode     acode;
    Record      record;
```

```
    Integer    input_fifo, loack_fifo, lotcpip_socket;
    PumpInfo   pump_info;
    Boolean    have_data;
loack_fifo   = OpenFile ( LOACK_FIFO );
input_fifo   = OpenFile ( INPUT_FIFO );
lotcpip_socket = OpenSocket ( LOTCP_SOCKET );
ReadFile ( input_fifo, &pump_info, sizeof(pump_info) );
DoForever
    AwaitConnection ( lotcpip_socket );
    WriteFile ( lotcpip_socket, &pump_info, sizeof(pump_info) );
    acode = ACK_OK;
    have_data = FALSE;
    DoWhile ( Connected(lotcpip_socket) )
        If ( acode == ACK_OK ) Then
            If ( have_data == TRUE ) Then
                WriteFile ( lotcpip_socket, &acode, sizeof(AckCode) );
                pump_info.record = record;
            EndIf
            ReadFile ( lotcpip_socket, &record.hdr.reclen, sizeof(Integer) );
            ReadFile ( lotcpip_socket, record.dta, record.hdr.reclen );
            acode = ACK_RESEND;
            have_data = TRUE;
        Else
            WriteFile ( input_fifo, &record.hdr.reclen, sizeof(Integer) );
            WriteFile ( input_fifo, record.dta, record.hdr.reclen );
            ReadFile ( loack_fifo, &acode, sizeof(AckCode) );
        EndIf
    EndWhile
EndForever
```

4 Configuration

In this section we discuss the parameter and configuration files used to configure the Event Driven Pump. The /etc/pump_param file contains parameter definitions used to modify the default behavior of the Event Driven Pump. It contains entries of the form <param>=<value>. All values must be either TEXT or INTEGER depending upon <param>. The following are the parameters with their default values shown:

TIMER_INTERVAL=8

> The granularity of the two timers used by the Pump. There are 7812 time units in a second, so 8 is roughly a millisecond. Decreasing this value may degrade global system performance as the timers approach constant looping with no yielding of CPU cycles. Increasing this value may degrade the Pumps performance. All other parameters that make reference to time will use TIMER_INTERVAL as the basic unit of time.

ACK_TIMEOUT=50

> This is the number of TIMER_INTERVALS to expire before which the Ack module must send an Ack otherwise the Ack module will send a NACK. See NRL Pump paper for more details.

EXEC_PATH=/epump/bin

> This is where the ring-3 Pump executables (s2f & f2s) reside.

FILE_PATH=/epump/files

> This is where the FIFO files are located.

IPC_INTERVAL=500

> This parameter specifies the number of TIMER_INTERVALS to wait between inter-module communication attempts. See IPC_RETRIES below.

IPC_RETRIES=1000

> It is possible during the Pump's startup process for one module to attempt communication with another module which is not ready for communications yet. This parameter specifies how many times an attempt should be made before aborting.

MOVAVG_EPSILON=500

> This number is divided by 1000, yielding .0625ms. This is the smallest allowed mean for the modified exponential distribution (this prevents a singularity at zero). See [3] for details.

MOVAVG_ORDER=50

> This is the number of transactions over which the moving average will be taken when determining high's rate of delivery.

MOVAVG_SEED=5

> This is the initial moving average rate of high delivery. Should have little affect on the pump unless it is set very high.

SAFB_PATH=/usr/local/epump

> This is where the Store And Forward Buffer (SAFB) file will reside. It will be created by the Pump and will be approx QUESIZE * RECSIZE bytes in length.

SAFB_QUESIZE=50

> This is the maximum number of transactions that can be pending within the Pump. This default value represents the maximum possible with STOP 4.1 due to IPC_MESSAGE queuing restrictions.

SAFB_RECSIZE=65535

> This is the maximum bytes of data that can be processed for a single transaction by the Pump. This default value represents the maximum possible with STOP 4.1 due to FIFO restrictions.

The `/etc/pump_config` contains Pump definitions. A Pump definition must specify a low and high *(host, port)* pair to define the low and high side of each Pump. The *host* must be a valid host name found in the `/etc/hosts` file, and the *port* must be an unused port number or name found in the `/etc/services` file. An example Pump definition might be: `genser,pump0-in:sci,pump0-out`, where (`genser,pump0-in`) are the *(host, port)* on the low side and (`sci,pump0-out`) are the *(host, port)* on the high side.

5 Interface

In this section we present a definition of the Event Driven Pump's Interface. Any application which intends to use the Event Driven Pump must adhere to the following protocol:

1. A client application must initiate a socket connection on the port allocated for Pump communication.

2. After a connection is established the following information will be sent to the client application by the Event Driven Pump:

   - Major version of Event Driven Pump (1 byte)
   - Minor version of Event Driven Pump (1 byte)
   - Ack timeout in milliseconds (4 bytes)
   - Connection timeout in seconds (4 bytes)
   - Maximum bytes for a message (4 bytes)
   - Last Acked message (2 byte length and then data)

3. All data transactions must be in the form of a message which is a 2 byte length field followed by the variable length message data. The length of the data message cannot exceed the value specified for the SAFB_RECSIZE parameter.

4. The Pump guarantees delivery of a message that it Acks with an AckCode of zero.

5. It is recommended that the client application include a block ID or similar bookkeeping field within the message so that duplicate messages can be recognized and handled appropriately.

6 Summary

We have presented arguments justifying the importance of balancing security and functional requirements for secure communication devices. A device which meets these requirements called the Basic Pump is presented and specified in some detail. An implementation of the Basic Pump called the Event Driven Pump was described and the pseudo code presented.

References

[1] D. Bell and L. LaPadula. "Secure computer systems: Mathematical Foundation," ESD-TR-73-278, Vol.1, Mitre Corp, 1973.

[2] M. H. Kang and I. S. Moskowitz. "A Pump for rapid, reliable, secure communication," Proceedings ACM Conf. Computer & Commun. Security '93, pp. 119 - 129, Fairfax, VA, 1993.

[3] M. H. Kang and I. S. Moskowitz. "A data Pump for communication," Submitted for publication. *http://www.itd.nrl.navy.mil/ITD/5540/publications/CHACS/index1995.html*

[4] M.H. Kang, I. S. Moskowitz and D. Lee."A network version of the Pump," Proc. of the IEEE Symposium on Research in Security and Privacy, Oakland, Ca, May 1995.

[5] Myong H. Kang and I. S. Moskowitz, "A generalized Pump," In preparation.

[6] I. S. Moskowitz and M. H. Kang. "Covert channels — Here to stay?,"Proceedings COM-PASS '94, pp. 235 - 243, Gaithersburg, MD, 1994.

[7] I. S. Moskowitz and M. H. Kang. "The Modulated-Input Modulated-Output model," Proceedings IFIP WG 11.3 working conference on database security, Rensselaerville, NY, August 1995.

[8] R.S. Sandhu. "Lattice-based access control models," Computer (IEEE), Vol. 26, No. 11, pp. 9-19, Nov. 1993.